19/02/2024
Monday

* **Topological sort -**

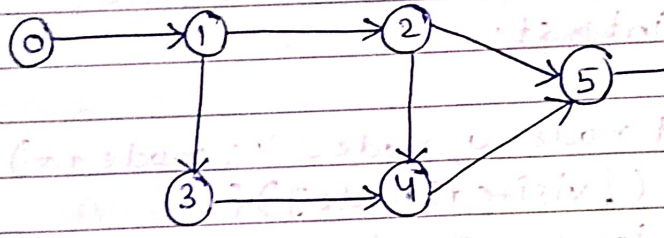
Is a linear ordering of vertices such that for edge $u \to v$, u comes before v in that ordering. For this, the graph must be a Directed Acyclic graph (DAG). Topological sort can be implemented by both dfs or bfs.

Topological sort using DFS -

- Do DFS traversal on graph.
- While backtracking, insert the node in stack.

hum stack me vhi node insert krenge jispr koi other node depend nhi kr rhi ya jo depend kr rhi thi vo process ho gyi.

Let the graph be -



adjacency list -
0 → {1}
1 → {2, 3}
2 → {4, 5}
3 → {4}
4 → {5}
5 → {6}

dfs(0)
dfs(1)
dfs(2) dfs(3) ~~dfs(4)~~ already visited
dfs(4) ~~dfs(5)~~ already visited
dfs(5)
dfs(6)

Stack -

0
1
3
2
4
5
6

visited -
0 → ~~F~~ T
1 → ~~F~~ T
2 → ~~F~~ T
3 → ~~F~~ T
4 → ~~F~~ T
5 → ~~F~~ T
6 → ~~F~~ T

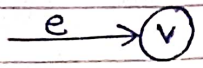
topological order -
0, 1, 3, 2, 4, 5, 6

Code -

```
class Solution {
public:

void topoSortDfs(int src, vector<int>adj[],
 unordered_map<int,bool>& visited, stack<int>&st)
{
    visited[src]= true;

    for(auto nbr: adj[src]){
        if(!visited[nbr]){
            topoSortDfs(nbr, adj, visited, st);
        }
    }
    st.push(src);
}
vector<int>topoSort(int V, vector<int>adj[]){
  unordered_map<int, bool> visited;
  stack<int> st;

  for(int node=0; node<V; node++){
      if(!visited[node]){
        topoSortDfs(node, adj, visited, st);
      }
  }

  vector<int>v;
  while(!st.empty()){
     v.push_back(st.top());
     st.pop();
  }
  return v;
}
};
```

Note - Topological sort mein sbse pehli node sbse independent node hoti hein jo kisi pr depend nhi kr rhi.

* **Topological sort using BFS - kahn's algorithm**



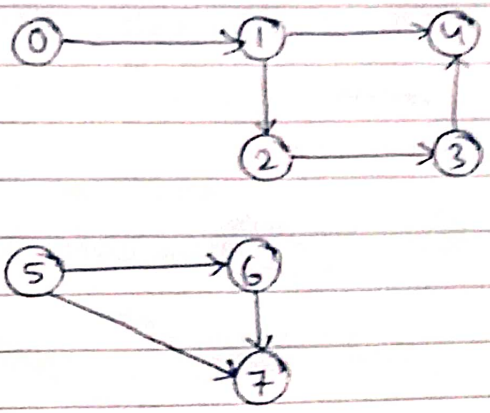
Jis node pr koi bhi incoming edge nhi hogi i.e. indegree 0 hogi, vo sbse independent node hogi. As we are using BFS here, so queue me hum sirf vhi node push krenge jiski indegree 0 hogi. for that reasons, hume hr node ki indegree calculate krni hogi. At the end, queue me jo ordering bnegi, that will be topological sorting.

Indegree of a node tells its dependency.

Logic -

(1) Do BFS traversal on adjacency list.
(2) Calculate indegree of the nodes in graph.
(3) Push all the nodes in queue with 0 indegree.
(4) While queue is not empty -
- fetch front node. Pop it and store it.
- Traverse on the neighbouring nodes of front node. Decrement their indegree by one. If indegree gets zero, then push that neighbour node in queue.

let the graph be -



adjacency list -

0 → {1}
1 → {2, 4}
2 → {3}
3 → {4}
4 → { }
5 → {6, 7}
6 → {7}
7 → { }

indegree[node] -

0 → 0
1 → ✗ 0
2 → ✗ 0
3 → ✗ 0
4 → ✗ ✗ 0
5 → 0
6 → ✗ 0
7 → ✗ ✗ 0

To maintain initial state of queue, push the nodes in queue whom indegree equals 0.

queue → | 0 | 5 | * | * | * | * | * | * |

topological order - 0 5 1 6 2 7 3 4

Code-

```
void topoSortBfs(int n, vector<int>&v){

    queue<T>q;
    unordered_map<T, int>indegree;

    //calculate indegree
    for(auto i : adj){
        for(auto nbr: i.second){
            indegree[nbr]++;
        }
    }

    //push all the nodes with indegree = 0 in queue
    for(int i=0; i<n; i++){
        if(indegree[i]==0){
            q.push(i);
        }
    }

    //BFS logic
    while(!q.empty()){
        T frontNode = q.front();
        q.pop();
        v.push_back(frontNode);

        for(auto nbr: adj[frontNode]){
            indegree[nbr]--;

            //check if indegree = 0
            if(indegree[nbr]==0){
                q.push(nbr);
            }
        }
    }
}
```

Note - This algorithm can also be applied in directed graph.

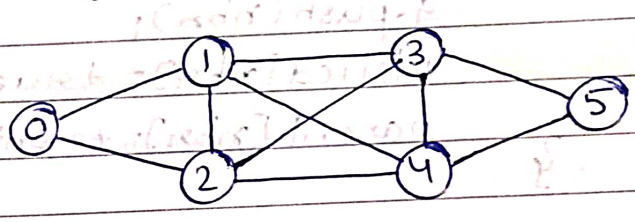
* **Shortest path in undirected graph using BFS -**

Given is source and destination node in a graph. Find the shortest path between the two.

for this, we have to track that which node approaches or visited the destination node for the first time. for that, we will keep a map visited so that no node is re-visited.

The main point here is to keep the list of parents so that we can track the path.

let the graph be -



adjacency list-
0 → {1, 2}
1 → {0, 2, 3, 4}
2 → {0, 1, 3, 4}
3 → {1, 2, 4, 5}
4 → {2, 1, 3, 5}
5 → {3, 4}

visited -
0 → ~~F~~ T
1 → ~~F~~ T
2 → ~~F~~ T
3 → ~~F~~ T
4 → ~~F~~ T
5 → ~~F~~ T

parent -
0 → -1
1 → 0
2 → 0
3 → 1
4 → 1
5 → 3

queue | ~~0~~ | ~~1~~ | ~~2~~ | ~~3~~ | ~~4~~ | ~~5~~ |

Now, we can see here, the shortest path is -

0 → 1 → 3 → 5

Code -

```
void shortestPath(T src, T dest){
    queue<T> q;
    unordered_map<T, bool> visited;
    unordered_map<T, T> parent;

    //initial state
    q.push(src);
    visited[src] = true;
    parent[src] = -1;

    while(!q.empty()){
        T frontNode = q.front();
        q.pop();

        for(auto nbr : adj[frontNode]){
            if(!visited[nbr]){
                q.push(nbr);
                visited[nbr] = true;
                parent[nbr] = frontNode;
            }
        }
    }
//this vector will store the shortest path
    vector<int> v;
    while(dest != -1){
        v.push_back(dest);
        dest = parent[dest];
    }
    reverse(v.begin(), v.end());
}
```